Database System Implementation

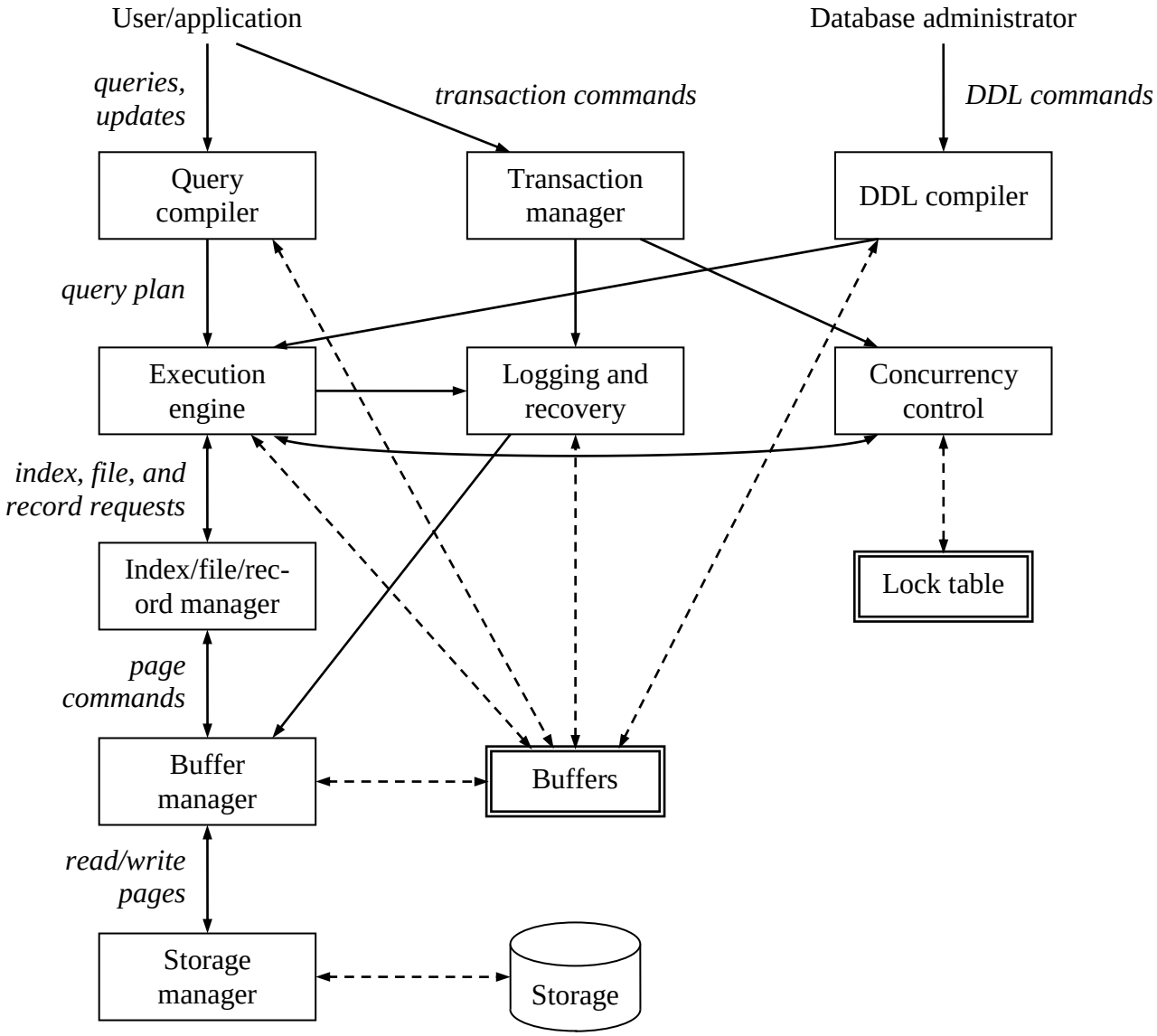
Textbook: Hector Garcia-Molina – Jeffrey D. Ullman – Jennifer Widom: *Database Systems – The Complete Book*, Second Edition, Pearson Prentice Hall, 2009, Chapters 17 and 18

Prerequisite: Database Systems course.

Topics: System failures and logging techniques against them; concurrency control.

Introduction

Database Management System Components



The figure illustrates the architecture of a general database management system (DBMS). Single-lined boxes denote the components of the system, whereas double-lined boxes represent in-memory data structures. Solid arrows denote control flow accompanied by data flow, and dashed arrows denote only data flow. The great majority of interactions with the DBMS follow the path on the left side of the figure. A user

or an application program initiates some action, using the data manipulation language (DML). This command does not affect the schema of the database but may affect the content of the database (if the action is a modification command) or will extract data from the database (if the action is a query). DML statements are handled by two separate subsystems:

1. *Answering the query.* The query is parsed and optimized by a query compiler. The resulting query execution plan (query plan for short), or sequence of actions the DBMS will perform to answer the query, is passed to the execution engine. The execution engine issues a sequence of requests for small pieces of data, typically records or tuples of a relation, to a resource manager that knows about data files (holding relations), the format and size of records in those files, and index files, which help find elements of data files quickly. The requests for data are passed to the buffer manager. The buffer manager's task is to bring appropriate portions of the data from secondary storage (disk) where it is kept permanently, to the main-memory buffers. Normally, the page or "disk block" is the unit of transfer between buffers and disk. The buffer manager communicates with a storage manager to get data from disk. The storage manager might involve operating-system commands, but more typically, the DBMS issues commands directly to the disk controller.
2. *Transaction Processing.* Queries and other DML actions are grouped into transactions, which are units that must be executed atomically and in isolation from one another. Any query or modification action can be a transaction by itself. In addition, the execution of transactions must be durable, meaning that the effect of any completed transaction must be preserved even if the system fails in some way right after completion of the transaction. We divide the transaction processor into two major parts:
    a) *Concurrency control manager* or *scheduler*: responsible for assuring atomicity and isolation of transactions.
    b) *Logging and recovery manager*: responsible for the atomicity and durability of transactions.

Transaction

The transaction is the unit of execution of database operations, consisting of DML statements, and having the following properties:

- ***A****tomicity*: the all-or-nothing execution of transactions (the database operations in a transaction are either fully executed or not at all). Either all relevant data has to be changed in the database or none at all. This means that if one part of a transaction fails, the entire transaction fails, and the database state is left unchanged.
- ***C****onsistency preservation*: transactions are expected to preserve the consistency of the database, i.e., after the execution of a transaction, all consistency (or integrity) constraints (expectations about data elements and the relationships among them) defined in the database should be satisfied.
- ***I****solation*: the fact that each transaction must appear to be executed as if no other transaction were executing at the same time.
- ***D****urability*: the condition that the effect on the database of a transaction must never be lost, once the transaction has completed.

These are the *ACID properties* of transactions. From the DBMS's point of view, consistency preservation is always considered satisfied (see later: correctness principle), the other three properties, however, must be forced by the DBMS, although, sometimes we set aside some of them. For example, if we are issuing ad-hoc commands to a SQL system, then each query or database modification statement (plus any resulting trigger actions) is a transaction. When using an embedded SQL interface, the programmer controls the extent of a transaction, which may include several queries or modifications, as well as operations performed

in the host language. In the typical embedded SQL system, transactions begin as soon as operations on the database are executed and end with an explicit `COMMIT` or `ROLLBACK` ("abort") statement.

Transaction Processing

The transaction processor provides concurrent access to data and supports resilience (i.e., data integrity after a system failure) by executing transactions correctly. The transaction manager therefore accepts transaction commands from an application, which tell the transaction manager when transactions begin and end, as well as information about the expectations of the application (some may not wish to require atomicity, for example). The transaction processor performs the following tasks:

1. *Logging*: In order to assure durability, every change in the database is logged separately on disk. The log manager follows one of several policies designed to assure that no matter when a system failure or "crash" occurs, a recovery manager will be able to examine the log of changes and restore the database to some consistent state. The log manager initially writes the log in buffers and negotiates with the buffer manager to make sure that buffers are written to disk (where data can survive a crash) at appropriate times.
2. *Concurrency control*: Transactions must appear to execute in isolation. But in most systems, there will in truth be many transactions executing at once. Thus, the scheduler (concurrency control manager) must assure that the individual actions of multiple transactions are executed in such an order that the net effect is the same as if the transactions had in fact executed in their entirety, one-at-a-time. A typical scheduler does its work by maintaining locks on certain pieces of the database. These locks prevent two transactions from accessing the same piece of data in ways that interact badly. Locks are generally stored in a main-memory lock table, as suggested by the above figure. The scheduler affects the execution of queries and other database operations by forbidding the execution engine from accessing locked parts of the database.
3. *Deadlock resolution*: As transactions compete for resources through the locks that the scheduler grants, they can get into a situation where none can proceed because each needs something another transaction has. The transaction manager has the responsibility to intervene and cancel ("rollback" or "abort") one or more transactions to let the others proceed.

Coping with System Failures

This chapter deals with techniques for supporting the goal of *resilience*, that is, integrity of the data when the system fails in some way. (Data must not be corrupted simply because several error-free queries or database modifications are being done at once, either. This matter is addressed by concurrency control.)

The principal technique for supporting resilience is a *log*, which records securely the history of database changes. We shall discuss three different styles of logging, called "undo," "redo," and "undo/redo." We also discuss *recovery*, the process whereby the log is used to reconstruct what has happened to the database when there has been a failure. An important aspect of logging and recovery is avoidance of the situation where the log must be examined into the distant past. Thus, we shall learn about "*checkpointing*," which limits the length of log that must be examined during recovery.

We also discuss "*archiving*," which allows the database to survive not only temporary system failures but situations where the entire database is lost. Then, we must rely on a recent copy of the database (the archive) plus whatever log information survives to reconstruct the database as it existed at some point in the recent past. Finally, we shall learn about Oracle's logging and recovery management.

Failure Modes

There are many things that can go wrong as a database is queried and modified. Problems range from the keyboard entry of incorrect data to an explosion in the room where the database is stored on disk. The following items are a catalog of the most important failure modes and what the DBMS can do about them.

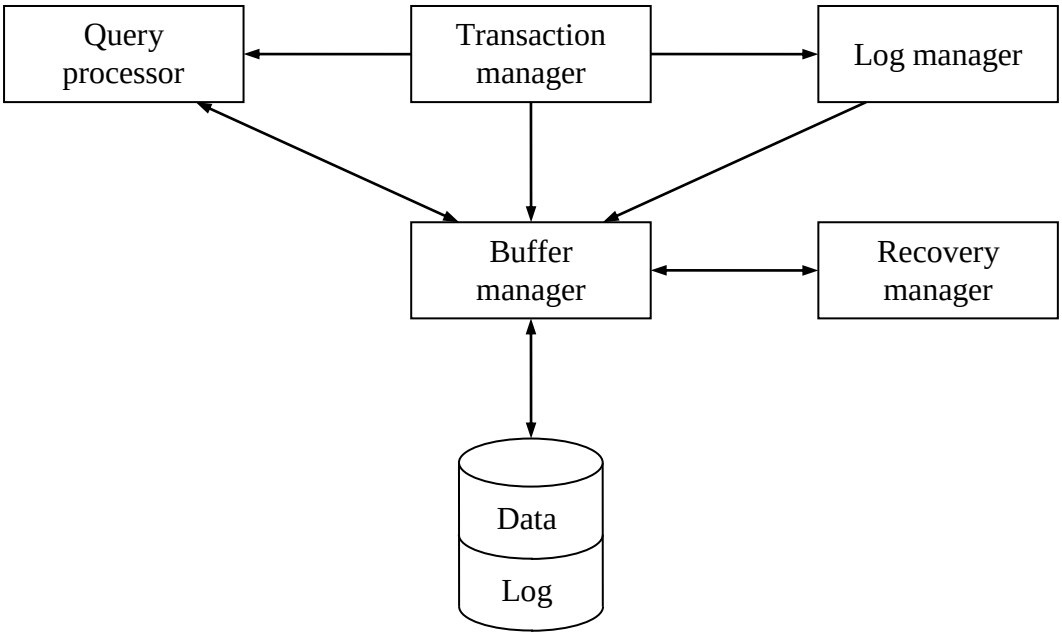
- *Erroneous data entry*: Some data errors are impossible to detect. For example, if a clerk mistypes one digit of your phone number, the data will still look like a phone number that could be yours. On the other hand, if the clerk omits a digit from your phone number, then the data is evidently in error, since it does not have the form of a phone number. The principal technique for addressing data entry errors is to write constraints and triggers that detect data believed to be erroneous. Triggers are program codes that execute automatically, typically in case of modifications of a certain type (such as inserting a row into a relation) in order to check if the new data satisfy the constraints defined by the designer of the database.
- *Media failures*: A local failure of a disk, one that changes only a bit or a few bits, can normally be detected by parity checks associated with the sectors of the disk. Head crashes, where the entire disk becomes unreadable, are generally handled by one or more of the following approaches:
    1. Use one of the *RAID* (Redundant Array of Independent Disks) schemes, so the lost disk can be restored.
    2. Maintain an *archive*, a copy of the database on a medium such as tape or optical disk. The archive is periodically created, either fully or incrementally, and stored at a safe distance from the database itself.
    3. Instead of an archive, one could keep *redundant copies of the database on-line*, distributed among several sites. In this case, consistency of the copies must be enforced.
- *Catastrophic failures*: In this category are a number of situations in which the media holding the database is completely destroyed. Examples include explosions, fires, or vandalism at the site of the database, as well as viruses. RAID will not help, since all the data disks and their parity check disks become useless simultaneously. However, the other approaches that can be used to protect against media failure — archiving and redundant, distributed copies — will also protect against a catastrophic failure.
- *System failures*: Each transaction has a *state*, which represents what has happened so far in the transaction. The state includes the current place in the transaction's code being executed and the values of any local variables of the transaction that will be needed later on. System failures are problems that cause the state of a transaction to be lost. Typical system failures are power loss and software errors. Since main memory is "volatile," a power failure will cause the contents of main memory to disappear, along with the result of any transaction step that was kept only in main memory, rather than on (nonvolatile) disk. Similarly, a software error may overwrite part of main memory, possibly including values that were part of the state of the program. When main memory is lost, the transaction state is lost; that is, we can no longer tell what parts of the transaction, including its database modifications, were made. Running the transaction again may not fix the problem. For example, if the transaction must add 1 to a value in the database, we do not know whether to repeat the addition of 1 or not. The principal remedy for the problems that arise due to a system error is logging of all database changes in a separate, nonvolatile log, coupled with recovery when necessary. However, the mechanisms whereby such logging can be done in a fail-safe manner are surprisingly intricate.

The Log Manager and the Transaction Manager

Assuring that transactions are executed correctly is the job of a *transaction manager*, a subsystem that performs several functions, including:

- issuing signals to the log manager (described below) so that necessary information in the form of "log records" can be stored on the log;
- assuring that concurrently executing transactions do not interfere with each other in ways that introduce errors (scheduling).

The transaction manager and its interactions are suggested by the following figure:



The transaction manager will send messages about actions of transactions to the log manager, to the buffer manager about when it is possible or necessary to copy the buffer back to disk, and to the query processor to execute the queries and other database operations that comprise the transaction.

The log manager maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times, these buffers must be copied to disk. The log, as well as the data, occupies space on the disk, as we suggest in the figure.

When there is a crash, the recovery manager is activated. It examines the log and uses it to repair the data if necessary. As always, access to the disk is through the buffer manager.

Correct Execution of Transactions

Before we can deal with correcting system errors, we need to understand what it means for a transaction to be executed "correctly." To begin, we assume that the database is composed of "elements." A *database element* is a functional unit of data stored in the physical database, whose value can be read or updated by transactions. A relation (or its object-oriented counterpart, a class extent), a tuple (or its OO counterpart, an object), or a disk block (or page) can all be considered as a database element. However, there are several good reasons in practice to use disk blocks or pages as the database element. In this way, buffer contents become single elements, allowing us to avoid some serious problems with logging and transactions. Avoiding database elements that are bigger than disk blocks also prevents a situation where part but not all of an element has been placed in nonvolatile storage (disk) when a crash occurs.

A database has a *state*, which is a value for each of its elements. Intuitively, we regard certain states as *consistent*, and others as *inconsistent*. Consistent states satisfy all constraints of the database schema, such as key constraints and constraints on values. *Explicit constraints* are enforced by the database, so any transaction that violates them will be rejected by the system and not change the database at all. However, consistent states must also satisfy *implicit constraints* that are in the mind of the database designer. The implicit constraints may be maintained by triggers that are part of the database schema, but they might also be maintained only by policy statements concerning the database, or warnings associated with the user interface through which updates are made. Implicit constraints cannot be characterized exactly under any circumstances. Our position is that if someone is given authority to modify the database, then they also have the authority to judge what the implicit constraints are.

A fundamental assumption about transactions is the *correctness principle*: If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transaction ends (isolation + atomicity –> consistency preservation). There is a converse to the correctness principle that forms the motivation for both the logging techniques and the concurrency control mechanisms. This converse involves two points:

- A transaction is atomic; that is, it must be executed as a whole or not at all. If only part of a transaction executes, then the resulting database state may not be consistent.
- Transactions that execute simultaneously are likely to lead to an inconsistent state unless we take steps to control their interactions.

The Primitive Operations of Transactions

Let us now consider in detail how transactions interact with the database. There are three address spaces that interact in important ways:

1. the space of disk blocks holding the database elements;
2. the virtual or main memory address space that is managed by the buffer manager;
3. the local address space of the transaction.

For a transaction to read a database element, that element must first be brought to a main-memory buffer or buffers, if it is not already there. Then, the contents of the buffer(s) can be read by the transaction into its own address space. Writing a new value for a database element by a transaction follows the reverse route. The new value is first created by the transaction in its own space. Then, this value is copied to the appropriate buffer(s). It is very important to see that transactions may not write a new value for a database element directly on the disk.

The buffer may or may not be copied to disk immediately; that decision is the responsibility of the buffer manager in general. As we shall soon see, one of the principal tools for assuring resilience is forcing the buffer manager to write the block in a buffer back to disk at appropriate times. However, in order to reduce the number of disk I/O's, database systems can and will allow a change to exist only in volatile main-memory storage, at least for certain periods of time and under the proper set of conditions.

In order to study the details of logging algorithms and other transaction management algorithms, we need a notation that describes all the operations that move data between address spaces. The primitives we shall use are:

1. `INPUT(X)`: Copy the disk block containing database element `X` to a memory buffer.
2. `READ(X,t)`: Copy the database element `X` to the transaction's local variable `t`. More precisely, if the block containing database element `X` is not in a memory buffer, then first execute `INPUT(X)`. Next, assign the value of `X` to local variable `t`.

3. `WRITE(X,t)`: Copy the value of local variable `t` to database element `X` in a memory buffer. More precisely, if the block containing database element `X` is not in a memory buffer, then execute `INPUT(X)`. Next, copy the value of `t` to `X` in the buffer.
4. `OUTPUT(X)`: Copy the block containing `X` from its buffer to disk.

The above operations make sense as long as database elements reside within a single disk block and therefore within a single buffer. If a database element occupies several blocks, we shall imagine that each block-sized portion of the element is an element by itself. The logging mechanism to be used will assure that the transaction cannot complete without the write of `X` being atomic; i.e., either all blocks of `X` are written to disk, or none are. Thus, we shall assume for the entire discussion of logging that a database element is no larger than a single block.

Different DBMS components issue the various commands we just introduced. `READ` and `WRITE` are issued by transactions. `INPUT` and `OUTPUT` are normally issued by the buffer manager. `OUTPUT` can also be initiated by the log manager under certain conditions, as we shall see.

Example. To see how the above primitive operations relate to what a transaction might do, let us consider a database that has two elements, `A` and `B`, with the constraint that they must be equal in all consistent states. Transaction `T` consists logically of the following two steps:

```
A := A*2;
B := B*2;
```

If `T` starts in a consistent state (i.e., `A` = `B`) and completes its activities without interference from another transaction or system error, then the final state must also be consistent. That is, `T` doubles two equal elements to get new, equal elements.

Execution of `T` involves reading `A` and `B` from disk, performing arithmetic in the local address space of `T`, and writing the new values of `A` and `B` to their buffers. The relevant steps of `T` are thus:

```
READ(A,t); t := t*2; WRITE(A,t);
READ(B,t); t := t*2; WRITE(B,t);
```

In addition, the buffer manager will eventually execute the `OUTPUT` steps to write these buffers back to disk. The following table shows the primitive steps of `T`, followed by the two `OUTPUT` commands from the buffer manager. We assume that initially `A` = `B` = 8. The values of the memory and disk copies of `A` and `B` and the local variable `t` in the address space of transaction `T` are indicated for each step:

Action	*t*	*M*–A	*M*–B	*D*–A	*D*–B
`READ(A,t)`	8	8		8	8
`t := t*2`	16	8		8	8
`WRITE(A,t)`	16	16		8	8
`READ(B,t)`	8	16	8	8	8
`t := t*2`	16	16	8	8	8
`WRITE(B,t)`	16	16	16	8	8
`OUTPUT(A)`	16	16	16	16	8
`OUTPUT(B)`	16	16	16	16	16

At the first step, `T` reads `A`, which generates an `INPUT(A)` command for the buffer manager if `A`'s block is not already in a buffer. The value of `A` is also copied by the `READ` command into local variable `t` of `T`'s address space. The second step doubles `t`; it has no effect on `A`, either in a buffer or on disk. The third step writes `t` into `A` in the buffer; it does not affect `A` on disk. The next three steps do the same for `B`, and the last two steps copy `A` and `B` to disk.

Observe that as long as all these steps execute, consistency of the database is preserved. If a system error occurs before `OUTPUT(A)` is executed, then there is no effect to the database stored on disk; it is as if `T`

never ran, and consistency is preserved. However, if there is a system error after `OUTPUT(A)` but before `OUTPUT(B)`, then the database is left in an inconsistent state. We cannot prevent this situation from ever occurring, but we can arrange that when it does occur, the problem can be repaired — either both `A` and `B` will be reset to 8, or both will be advanced to 16.

Example. Suppose that the consistency constraint on the database is $0 \leq A \leq B$. Tell whether each of the following transactions preserves consistency.

a) `A := A + B; B := A + B;`
b) `B := A + B; A := A + B;`
c) `A := B + 1; B := A + 1;`

Undo Logging

A *log* is a file of *log records*, each telling something about what some transaction has done. If log records appear in nonvolatile storage, we can use them to restore the database to a consistent state after a system crash. Our first style of logging — *undo logging* — makes repairs to the database state by undoing the effects of transactions that may not have completed before the crash.

Additionally, in this chapter we introduce the basic idea of log records, including the commit (successful completion of a transaction) action and its effect on the database state and log. We shall also consider how the log itself is created in main memory and copied to disk by a "flush-log" operation. Finally, we examine the undo log specifically, and learn how to use it in recovery from a crash. In order to avoid having to examine the entire log during recovery, we introduce the idea of "checkpointing," which allows old portions of the log to be thrown away.

Log Records

Imagine the log as a file opened for appending only. As transactions execute, the log manager has the job of recording in the log each important event. One block of the log at a time is filled with log records, each representing one of these events. Log blocks are initially created in main memory and are allocated by the buffer manager like any other blocks that the DBMS needs. The log blocks are written to nonvolatile storage on disk as soon as it is feasible.

There are several forms of log record that are used with each of the types of logging we discuss. These are:

1. `<START T>`: This record indicates that transaction `T` has begun.
2. `<COMMIT T>`: Transaction `T` has completed successfully and will make no more changes to database elements. Any changes to the database made by `T` should appear on disk. However, because we cannot control when the buffer manager chooses to copy blocks from memory to disk, we cannot in general be sure that the changes are already on disk when we see the `<COMMIT T>` log record. If we insist that the changes already be on disk, this requirement must be enforced by the log manager (as is the case for undo logging).
3. `<ABORT T>`: Transaction `T` could not complete successfully. If transaction `T` aborts, no changes it made can have been copied to disk, and it is the job of the transaction manager to make sure that such changes never appear on disk, or that their effect on disk is canceled if they do. We shall discuss the matter of repairing the effect of aborted transactions later. There can be several reasons for a transaction to abort. The simplest is when there is some error condition in the code of the transaction itself, e.g., an attempted division by zero. The DBMS may also abort a transaction for one of several reasons. For instance, a transaction may be involved in a deadlock, where it and one or more other transactions each

hold some resource that the other needs. Then, one or more transactions must be forced by the system to abort (see later).

4. <T,X,v>: This is the *update record*. The meaning of this record is: transaction T has changed database element X, and its former value was v. The change reflected by an update record normally occurs in memory, not disk; i.e., the log record is a response to a WRITE action into memory, not an OUTPUT action to disk. Notice also that an undo log does not record the new value of a database element, only the old value. As we shall see, should recovery be necessary in a system using undo logging, the only thing the recovery manager will do is cancel the possible effect of a transaction on disk by restoring the old value.

The Undo Logging Rules

An undo log is sufficient to allow recovery from a system failure, provided transactions and the buffer manager obey two rules:

U_1: If transaction T modifies database element X, then the log record of the form <T,X,v> must be written to disk *before* the new value of X is written to disk (*write-ahead logging* or WAL).

U_2: If a transaction commits, then its COMMIT log record must be written to disk only *after* all database elements changed by the transaction have been written to disk, but as soon thereafter as possible.

To summarize rules U_1 and U_2, material associated with one transaction must be written to disk in the following order:

1. the log records indicating changed database elements;
2. the changed database elements themselves;
3. the COMMIT log record.

However, the order of the first two steps applies to each database element individually, not to the group of update records for a transaction as a whole.

In order to force log records to disk, the log manager needs a flush-log command that tells the buffer manager to copy to disk any log blocks that have not previously been copied to disk or that have been changed since they were last copied. In sequences of actions, we shall show FLUSH LOG explicitly. The transaction manager also needs to have a way to tell the buffer manager to perform an OUTPUT action on a database element. We shall continue to show the OUTPUT action in sequences of transaction steps.

Example. Let us reconsider the previously investigated transaction in the light of undo logging. We shall expand our table to show the log entries and flush-log actions that have to take place along with the actions of transaction T:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

In line (1) of the table, transaction T begins. The first thing that happens is that the <START T> record is written to the log. Line (2) represents the read of A by T. Line (3) is the local change to t, which affects neither the database stored on disk nor any portion of the database in a memory buffer. Neither lines (2) nor (3) require any log entry, since they have no effect on the database.

Line (4) is the write of the new value of A to the buffer. This modification to A is reflected by the log entry <T,A,8>, which says that A was changed by T and its former value was 8. Note that the new value, 16, is not mentioned in an undo log.

Lines (5) through (7) perform the same three steps with B instead of A. At this point, T has completed and must commit. The changed A and B must migrate to disk, but in order to follow the two rules for undo logging, there is a fixed sequence of events that must happen.

First, A and B cannot be copied to disk until the log records for the changes are on disk. Thus, at step (8) the log is flushed, assuring that these records appear on disk. Then, steps (9) and (10) copy A and B to disk. The transaction manager requests these steps from the buffer manager in order to commit T.

Now, it is possible to commit T, and the <COMMIT T> record is written to the log, which is step (11). Finally, we must flush the log again at step (12) to make sure that the <COMMIT T> record of the log appears on disk. Notice that without writing this record to disk, we could have a situation where a transaction has committed, but for a long time a review of the log does not tell us that it has committed. That situation could cause strange behavior if there were a crash, because a transaction that appeared to the user to have completed long ago would then be undone and effectively aborted.

As we look at a sequence of actions and log entries like in the table above, it is tempting to imagine that these actions occur in isolation. However, the DBMS may be processing many transactions simultaneously. Thus, the four log records for transaction T may be interleaved on the log with records for other transactions. Moreover, if one of these transactions flushes the log, then the log records from T may appear on disk earlier than is implied by the flush-log actions. There is no harm if log records reflecting a database modification appear earlier than necessary. The essential policy for undo logging is that we don't write the <COMMIT T> record until the OUTPUT actions for T are completed.

A trickier situation occurs if two database elements A and B share a block. Then, writing one of them to disk writes the other as well. In the worst case, we can violate rule U_1 by writing one of these elements prematurely. It may be necessary to adopt additional constraints on transactions in order to make undo logging work. For instance, we might use a locking scheme where database elements are disk blocks, as described later, to prevent two transactions from accessing the same block at the same time. This and other problems that appear when database elements are fractions of a block motivate our suggestion that blocks be the database elements.

Recovery Using Undo Logging

Suppose now that a system failure occurs. It is possible that certain database changes made by a given transaction were written to disk, while other changes made by the same transaction never reached the disk. If so, the transaction was not executed atomically, and there may be an inconsistent database state. The recovery manager must use the log to restore the database to some consistent state.

First, we consider only the simplest form of recovery manager, one that looks at the entire log, no matter how long, and makes database changes as a result of its examination. Later, we consider a more sensible approach, where the log is periodically "checkpointed," to limit the distance back in history that the recovery manager must go.

The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions. If there is a log record <COMMIT T>, then by undo rule U_2 all changes made by transaction

T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

However, suppose that we find a <START T> record on the log but no <COMMIT T> or <ABORT T> record. Then there could have been some changes to the database made by T that were written to disk before the crash, while other changes by T either were not made, or were made in the main-memory buffers but not copied to disk. In this case, T is an *incomplete transaction* and must be undone. That is, whatever changes T made must be reset to their previous value. Fortunately, rule U_1 assures us that if T changed X on disk before the crash, then there will be a <T,X,v> record on the log, and that record will have been copied to disk before the crash. Thus, during the recovery, we must write the value v for database element X. Note that this rule raises the question whether X had value v in the database anyway; we don't even bother to check.

Since there may be several uncommitted transactions in the log, and there may even be several uncommitted transactions that modified X, we have to be systematic about the order in which we restore values. Thus, the recovery manager must scan the log from the end (i.e., from the most recently written record to the earliest written). As it travels, it remembers all those transactions T for which it has seen a <COMMIT T> record or an <ABORT T> record. Also as it travels backward, if it sees a record <T,X,v>, then:

- if T is a transaction whose COMMIT record has been seen, then do nothing, as T is committed and must not be undone (T is completed);
- if an ABORT record has been seen for transaction T, then again do nothing, as T has already been recovered (T is completed);
- otherwise, T is an incomplete transaction, so the recovery manager must change the value of X in the database to v, in case X had been altered just before the crash.

After making these changes, the recovery manager must write a log record <ABORT T> for each incomplete transaction T, and then flush the log. Now, normal operation of the database may resume, and new transactions may begin executing.

Example. Let us consider the sequence of actions from the above example. There are several different times that the system crash could have occurred; let us consider each significantly different one.

1. The crash occurs after step (12). Then the <COMMIT T> record reached disk before the crash. When we recover, we do not undo the results of T, and all log records concerning T are ignored by the recovery manager.
2. The crash occurs between steps (11) and (12). It is possible that the log record containing the COMMIT got flushed to disk; for instance, the buffer manager may have needed the buffer containing the end of the log for another transaction, or some other transaction may have asked for a log flush. If so, then the recovery is the same as in case (1) as far as T is concerned. However, if the COMMIT record never reached disk, then the recovery manager considers T incomplete. When it scans the log backward, it comes first to the record <T,B,8>. It therefore stores 8 as the value of B on disk. It then comes to the record <T,A,8> and makes A have value 8 on disk. Finally, the record <ABORT T> is written to the log, and the log is flushed.
3. The crash occurs between steps (10) and (11). Now, the COMMIT record surely was not written, so T is incomplete and is undone as in case (2).
4. The crash occurs between steps (8) and (10). Again, T is undone. In this case, the change to A and/or B may not have reached disk. Nevertheless, the proper value, 8, is restored for each of these database elements.

5. The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning T have reached disk. However, we know by rule U_1 that if the change to A and/or B reached disk, then the corresponding log record reached disk. Therefore, if there were changes to A and/or B made on disk by T, then the corresponding log record will cause the recovery manager to undo those changes.

Suppose the system again crashes while we are recovering from a previous crash. Because of the way undo log records are designed, giving the old value rather than, say, the change in the value of a database element, the recovery steps are idempotent, that is, repeating them many times has exactly the same effect as performing them once. We already observed that if we find a record <T,X,v>, it does not matter whether the value of X is already v — we may write v for X regardless. Similarly, if we repeat the recovery process, it does not matter whether the first recovery attempt restored some old values; we simply restore them again. The same reasoning holds for the other logging methods we discuss. Since the recovery operations are idempotent, we can recover a second time without worrying about changes made the first time.

Checkpointing

As we observed, recovery requires that the entire log be examined, in principle. When logging follows the undo style, once a transaction has its COMMIT log record written to disk, the log records of that transaction are no longer needed during recovery. We might imagine that we could delete the log prior to a COMMIT, but sometimes we cannot. The reason is that often many transactions execute at once. If we truncated the log after one transaction committed, log records pertaining to some other active transaction T might be lost and could not be used to undo T if recovery were necessary.

The simplest way to untangle potential problems is to *checkpoint* the log periodically. There are two kinds of checkpoints: simple and nonquiescent. In a simple checkpoint, we:

1. stop accepting new transactions;
2. wait until all currently active transactions commit or abort and have written a COMMIT or ABORT record on the log;
3. flush the log to disk;
4. write a log record <CKPT>, and flush the log again;
5. resume accepting transactions.

Any transaction that executed prior to the checkpoint will have finished, and by rule U_2, its changes will have reached the disk. Thus, there will be no need to undo any of these transactions during recovery. During a recovery, we scan the log backwards from the end, identifying incomplete transactions. However, when we find a <CKPT> record, we know that we have seen all the incomplete transactions. Since no transactions may begin until the checkpoint ends, we must have seen every log record pertaining to the incomplete transactions already. Thus, there is no need to scan prior to the <CKPT>, and in fact, the log before that point can be deleted or overwritten safely (unless it is needed for some other reason).

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<T_2,C,15>
<T_1,D,20>
<COMMIT T_1>
<COMMIT T_2>
<CKPT>
<START T_3>

<T_3,E,25>
<T_3,F,30>

Suppose we decide to do a checkpoint after the fourth entry. Since T_1 and T_2 are the active (incomplete) transactions, we shall have to wait until they complete before writing the <CKPT> record on the log. Suppose a crash occurs at the end of the listing. Scanning the log from the end, we identify T_3 as the only incomplete transaction and restore E and F to their former values 25 and 30, respectively. When we reach the <CKPT> record, we know there is no need to examine prior log records and the restoration of the database state is complete.

The question may arise how to find the last log record? It is common to recycle blocks of the log file on disk, since checkpoints allow us to drop old portions of the log. However, if we overwrite old log records, then we need to keep a serial number, which may only increase, as suggested by the following figure:

~~1~~	~~2~~	~~3~~	4	5	6	7	8
9	10	11					

Then, we can find the record whose serial number is greater than that of the next record; this record will be the current end of the log, and the entire log is found by ordering the current records by their present serial numbers. In practice, a large log may be composed of many files, with a "top" file whose records indicate the files that comprise the log. Then, to recover, we find the last record of the top file, go to the file indicated, and find the last record there.

Nonquiescent Checkpointing

A problem with the checkpointing technique described above is that effectively we must shut down the system while the checkpoint is being made. Since the active transactions may take a long time to commit or abort, the system may appear to users to be stalled. Thus, a more complex technique known as *nonquiescent checkpointing*, which allows new transactions to enter the system during the checkpoint, is usually preferred. The steps in a nonquiescent checkpoint are:

1. Write a log record <START CKPT(T_1, …, T_k)> and flush the log. Here, T_1, …, T_k are the names or identifiers for all the active transactions (i.e., transactions that have not yet committed and written their changes to disk).
2. Wait until all of T_1, …, T_k commit or abort, but do not prohibit other transactions from starting.
3. When all of T_1, …, T_k have completed, write a log record <END CKPT> and flush the log.

With a log of this type, we can recover from a system crash as follows. As usual, we scan the log from the end, finding all incomplete transactions as we go, and restoring old values for database elements changed by these transactions. There are two cases, depending on whether, scanning backwards, we first meet an <END CKPT> record or a <START CKPT(T_1, …, T_k)> record:

- If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous <START CKPT(T_1, …, T_k)> record. We may thus scan backwards as far as the next START CKPT and then stop; previous log is useless and may as well have been discarded.
- If we first meet a record <START CKPT(T_1, …, T_k)>, then the crash occurred during the checkpoint. However, the only incomplete transactions are those we met scanning backwards before we reached the START CKPT and those of T_1, …, T_k that did not complete before the crash. Thus, we need scan no further back than the start of the earliest of these incomplete transactions. The previous START CKPT record with a corresponding END CKPT is certainly prior to any of these transaction starts, but often we shall find the starts of the incomplete transactions long before we reach the previous checkpoint. If the previous START CKPT has no corresponding END CKPT record, then it means that another crash also occurred during a checkpoint. Such incomplete checkpoints must be ignored. Moreover, if we use

pointers to chain together the log records that belong to the same transaction, then we need not search the whole log for records belonging to active transactions; we just follow their chains back through the log.

As a general rule, once an <END CKPT> record has been written to disk, we can delete the log prior to the previous START CKPT record.

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<T_2,B,10>
<START CKPT(T_1,T_2)>
<T_2,C,15>
<START T_3>
<T_1,D,20>
<COMMIT T_1>
<T_3,E,25>
<COMMIT T_2>
<END CKPT>
<T_3,F,30>

Now, we decide to do a nonquiescent checkpoint after the fourth entry. Since T_1 and T_2 are the active (incomplete) transactions at this time, we write the fifth log record. Suppose that while waiting for T_1 and T_2 to complete, another transaction, T_3, initiates.

Suppose that at the end of the listing, there is a system crash. Examining the log from the end, we find that T_3 is an incomplete transaction and must be undone. The final log record tells us to restore database element F to the value 30. When we find the <END CKPT> record, we know that all incomplete transactions began after the previous START CKPT. Scanning further back, we find the record <T_3,E,25>, which tells us to restore E to value 25. Between that record and the START CKPT, there are no other transactions that started but did not commit, so no further changes to the database are made.

Now suppose the crash occurs during the checkpoint, and the end of the log after the crash is the <T_3,E,25> record. Scanning backwards, we identify T_3 and then T_2 as incomplete transactions and undo changes they have made. When we find the <START CKPT(T_1,T_2)> record, we know that the only other possible incomplete transaction is T_1. However, we have already scanned the <COMMIT T_1> record, so we know that T_1 is not incomplete. Also, we have already seen the <START T_3> record. Thus, we need only to continue backwards until we meet the START record for T_2, restoring database element B to value 10 as we go.

Redo Logging

Undo logging has a potential problem that we cannot commit a transaction without first writing all its changed data to disk. Sometimes, we can save disk I/O's if we let changes to the database reside only in main memory for a while. As long as there is a log to fix things up in the event of a crash, it is safe to do so.

The requirement for immediate backup of database elements to disk can be avoided if we use a logging mechanism called *redo logging*. The principal differences between redo and undo logging are:

- While undo logging cancels the effect of incomplete transactions and ignores committed ones during recovery, redo logging ignores incomplete transactions and repeats the changes made by committed transactions.
- While undo logging requires us to write changed database elements to disk before the COMMIT log record reaches disk, redo logging requires that the COMMIT record appear on disk before any changed values reach disk.
- While the old values of changed database elements are exactly what we need to recover when the undo rules U_1 and U_2 are followed, to recover using redo logging, we need the new values instead.

The Redo Logging Rule

In redo logging, an update log record is formally the same as in undo logging: <T,X,v>, but here it means "transaction T wrote new value v for database element X." There is no indication of the old value of X in this record. Every time a transaction T modifies a database element X, a record of the form <T,X,v> must be written to the log.

For redo logging, the order in which data and log entries reach disk can be described by a single "redo rule:"

R_1: Before modifying any database element X on disk, it is necessary that all log records pertaining to this modification of X, including both the update record <T,X,v> and the <COMMIT T> record, must appear on disk.

The COMMIT record for a transaction can only be written to the log when the transaction completes, so the commit record must follow all the update log records. Thus, when redo logging is in use, the order in which material associated with one transaction gets written to disk is:

1. the log records indicating changed database elements;
2. the COMMIT log record;
3. the changed database elements themselves.

Example. Let us consider transaction T defined earlier using redo logging:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

The major differences between using undo and redo logging are as follows. First, we note in lines (4) and (7) that the log records reflecting the changes have the new values of A and B, rather than the old values. Second, we see that the <COMMIT T> record comes earlier, at step (8). Then, the log is flushed, so all log records involving the changes of transaction T appear on disk. Only then can the new values of A and B be written to disk. We show these values written immediately, at steps (10) and (11), although in practice, they might occur later.

Recovery with Redo Logging

An important consequence of the redo rule R_1 is that unless the log has a <COMMIT T> record, we know that no changes to the database made by transaction T have been written to disk. Thus, incomplete transactions may be treated during recovery as if they had never occurred. However, the committed transactions present a problem, since we do not know which of their database changes have been written to disk. Fortunately, the redo log has exactly the information we need: the new values, which we may write to disk regardless of whether they were already there. To recover, using a redo log, after a system crash, we do the following:

1. Identify the committed transactions.
2. Scan the log forward from the beginning. For each log record <T,X,v> encountered:
   a) If T is not a committed transaction, do nothing.
   b) If T is committed, write value v for database element X.
3. For each incomplete transaction T, write an <ABORT T> record to the log and flush the log.

Example. Let us consider the log written in the above example and see how recovery would be performed if the crash occurred after different steps in that sequence of actions:

1. If the crash occurs any time after step (9), then the <COMMIT T> record has been flushed to disk. The recovery system identifies T as a committed transaction. When scanning the log forward, the log records <T,A,16> and <T,B,16> cause the recovery manager to write values 16 for A and B. Notice that if the crash occurred between steps (10) and (11), then the write of A is redundant, but the write of B had not occurred and changing B to 16 is essential to restore the database state to consistency. If the crash occurred after step (11), then both writes are redundant but harmless.

2. If the crash occurs between steps (8) and (9), then although the record <COMMIT T> was written to the log, it may not have gotten to disk (depending on whether the log was flushed for some other reason). If it did get to disk, then the recovery proceeds as in case (1), and if it did not get to disk, then recovery is as in case (3), below.

3. If the crash occurs prior to step (8), then <COMMIT T> surely has not reached disk. Thus, T is treated as an incomplete transaction. No changes to A or B on disk are made on behalf of T, and eventually an <ABORT T> record is written to the log.

Since several committed transactions may have written new values for the same database element X, we have required that during a redo recovery, we scan the log from earliest to latest. Thus, the final value of X in the database will be the one written last, as it should be. Similarly, when describing undo recovery, we required that the log be scanned from latest to earliest. Thus, the final value of X will be the value that it had before any of the incomplete transactions changed it.

Checkpointing a Redo Log

Redo logs present a checkpointing problem that we do not see with undo logs. Since the database changes made by a committed transaction can be copied to disk much later than the time at which the transaction commits, we cannot limit our concern to transactions that are active at the time we decide to create a checkpoint. Regardless of whether the checkpoint is quiescent or nonquiescent, between the start and end of the checkpoint, we must write to disk all database elements that have been modified by committed transactions. To do so requires that the buffer manager keep track of which buffers are *dirty*, that is, they have been changed but not written to disk. It is also required to know which transactions modified which buffers.

On the other hand, we can complete the checkpoint without waiting for the active transactions to commit or abort, since they are not allowed to write their pages to disk at that time anyway. The steps to perform a nonquiescent checkpoint of a redo log are as follows:

1. Write a log record <START CKPT(T_1, …, T_k)>, where T_1, …, T_k are all the active (uncommitted) transactions, and flush the log.
2. Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log.
3. Write an <END CKPT> record to the log and flush the log.

Example. Consider the following log:

<START T_1>
<T_1,A,5>
<START T_2>
<COMMIT T_1>
<T_2,B,10>
<START CKPT(T_2)>
<T_2,C,15>
<START T_3>
<T_3,D,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

When we start the checkpoint, only T_2 is active, but the value of A written by T_1 may have reached disk. If not, then we must copy A to disk before the checkpoint can end. We suggest the end of the checkpoint occurring after several other events have occurred: T_2 wrote a value for database element C, and a new transaction T_3 started and wrote a value of D. After the end of the checkpoint, the only things that happen are that T_2 and T_3 commit.

Recovery with a Checkpointed Redo Log

As for an undo log, the insertion of records to mark the start and end of a checkpoint helps us limit our examination of the log when a recovery is necessary. Also as with undo logging, there are two cases, depending on whether the last checkpoint record is START or END:

- Suppose first that the last checkpoint record on the log before a crash is <END CKPT>. Now, we know that every value written by a transaction that committed before the corresponding <START CKPT(T_1, …, T_k)> has had its changes written to disk, so we need not concern ourselves with recovering the effects of these transactions. However, any transaction that is either among the T_i's or that started after the beginning of the checkpoint can still have changes it made not yet migrated to disk, even though the transaction has committed. Thus, we must perform recovery as described earlier but may limit our attention to the transactions that are either one of the T_i's mentioned in the last <START CKPT(T_1, …, T_k)> or that started after that log record appeared in the log. In searching the log, we do not have to look further back than the earliest of the <START T_i> records. Notice, however, that these START records could appear prior to any number of checkpoints. Linking backwards all the log records for a given transaction helps us to find the necessary records, as it did for undo logging.
- Now, suppose the last checkpoint record on the log is <START CKPT(T_1, …, T_k)>. We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk. Thus, we must search back to the previous <END CKPT> record, find its matching

<START CKPT(S_1,...,S_m)> record, and redo all those committed transactions that either started after that START CKPT or are among the S_i's.

Example. Consider again the log from the previous example. If a crash occurs at the end, we search backwards, finding the <END CKPT> record. We thus know that it is sufficient to consider as candidates to redo all those transactions that either started after the <START CKPT(T_2)> record was written or that are on its list (i.e., T_2). Thus, our candidate set is (T_2, T_3). We find the records <COMMIT T_2> and <COMMIT T_3>, so we know that each must be redone. We search the log as far back as the <START T_2> record, and find the update records <T_2,B,10>, <T_2,C,15>, and <T_3,D,20> for the committed transactions. Since we don't know whether these changes reached disk, we rewrite the values 10, 15, and 20 for B, C, and D, respectively.

Now, suppose the crash occurred between the records <COMMIT T_2> and <COMMIT T_3>. The recovery is similar to the above, except that T_3 is no longer a committed transaction. Thus, its change <T_3,D,20> must not be redone, and no change is made to D during recovery, even though that log record is in the range of records that is examined. Also, we write an <ABORT T_3> record to the log after recovery.

Finally, suppose that the crash occurs just prior to the <END CKPT> record. In principal, we must search back to the next-to-last START CKPT record (with a corresponding <END CKPT>) and get its list of active transactions. However, in this case there is no previous checkpoint, and we must go all the way to the beginning of the log. Thus, we identify T_1 as the only committed transaction, redo its action <T_1,A,5>, and write records <ABORT T_2> and <ABORT T_3> to the log after recovery.

Since transactions may be active during several checkpoints, it is convenient to include in the <START CKPT(T_1,…,T_k)> records not only the names of the active transactions but pointers to the place on the log where they started. By doing so, we know when it is safe to delete early portions of the log. When we write an <END CKPT>, we know that we shall never need to look back further than the earliest of the <START T_i> records for the active transactions T_i. Thus, anything prior to that START record may be deleted.

Undo/Redo logging

We have seen two different approaches to logging, differentiated by whether the log holds old values or new values when a database element is updated. Each has certain drawbacks:

- Undo logging requires that data be written to disk immediately after a transaction finishes, perhaps increasing the number of disk I/O's that need to be performed.
- On the other hand, redo logging requires us to keep all modified blocks in buffers until the transaction commits and the log records have been flushed, perhaps increasing the average number of buffers required by transactions.
- Both undo and redo logs may put contradictory requirements on how buffers are handled during a checkpoint, unless the database elements are complete blocks or sets of blocks. For instance, if a buffer contains one database element A that was changed by a committed transaction and another database element B that was changed in the same buffer by a transaction that has not yet had its COMMIT record written to disk, then we are required to copy the buffer to disk because of A but also forbidden to do so, because rule R_1 applies to B.

We shall now see a kind of logging called *undo/redo logging*, which provides increased flexibility to order actions, at the expense of maintaining more information on the log.

The Undo/Redo Rules

An undo/redo log has the same sorts of log records as the other kinds of log, with one exception. The update log record that we write when a database element changes value has four components. Record <T,X,v,w> means that transaction T changed the value of database element X; its former value was v, and its new value is w. The constraints that an undo/redo logging system must follow are summarized by the following rule:

UR_1: Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update record <T,X,v,w> appear on disk.

Rule UR_1 for undo/redo logging thus enforces only the constraints enforced by both undo logging and redo logging. In particular, the <COMMIT T> log record can precede or follow any of the changes to the database elements on disk.

Example. Let us consider again transaction T defined earlier using undo/redo logging:

Step	*Action*	*t*	*M*–A	*M*–B	*D*–A	*D*–B	*Log*
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T,A,8,16>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T,B,8,16>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

Notice that the log records for updates now have both the old and the new values of A and B. In this sequence, we have written the <COMMIT T> log record in the middle of the output of database elements A and B to disk. Step (10) could also have appeared before step (8) or step (9), or after step (11).

Recovery with Undo/Redo Logging

When we need to recover using an undo/redo log, we have the information in the update records either to undo a transaction T by restoring the old values of the database elements that T changed, or to redo T by repeating the changes it has made. The undo/redo recovery policy is:

1. Redo all the committed transactions in the order earliest-first, and
2. undo all the incomplete transactions in the order latest-first.

Notice that it is necessary for us to do both. Because of the flexibility allowed by undo/redo logging regarding the relative order in which COMMIT log records and the database changes themselves are copied to disk, we could have either a committed transaction with some or all of its changes not on disk, or an uncommitted transaction with some or all of its changes on disk.

Example. Here are the different ways that recovery would take place on the assumption that there is a crash at various points in the sequence:

1. Suppose the crash occurs after the <COMMIT T> record is flushed to disk. Then T is identified as a committed transaction. We write the value 16 for both A and B to the disk. Because of the actual order of events, A already has the value 16, but B may not, depending on whether the crash occurred before or after step (11).

2. If the crash occurs prior to the <COMMIT T> record reaching disk, then T is treated as an incomplete transaction. The previous values of A and B, 8 in each case, are written to disk. If the crash occurs between steps (9) and (10), then the value of A was 16 on disk, and the restoration to value 8 is necessary. In this example, the value of B does not need to be undone, and if the crash occurs before step (9), then neither does the value of A. However, in general we cannot be sure whether restoration is necessary, so we always perform the undo operation.

Like undo logging, a system using undo/redo logging can exhibit a behavior where a transaction appears to the user to have been completed (e.g., they booked an airline seat over the Web and disconnected), and yet because the <COMMIT T> record was not flushed to disk, a subsequent crash causes the transaction to be undone rather than redone. If this possibility is a problem, we suggest the use of an additional rule for undo/redo logging:

*UR*2: A <COMMIT T> record must be flushed to disk as soon as it appears in the log.

For instance, we would add FLUSH LOG after step (10) in the example above.

You may have noticed that we did not specify whether undo's or redo's are done first during recovery using an undo/redo log. In fact, whether we perform the redo's or undo's first, we are open to the following situation: a transaction T has committed and is redone; however, T wrote a value X written also by some transaction U that has not committed and is undone. The problem is not whether we redo first, and leave X with its value prior to U, or we undo first and leave X with its value written by T. The situation makes no sense either way, because the final database state does not correspond to the effect of any sequence of atomic transactions.

In reality, the DBMS must do more than log changes. It must assure that such situations do not occur at all. We will later see a discussion about the means to isolate transactions like T and U, so the interaction between them through database element X cannot occur. We will explicitly address means for preventing this situation where T reads a "dirty" value of X — one that has not been committed.

Checkpointing an Undo/Redo Log

A nonquiescent checkpoint is somewhat simpler for undo/redo logging than for the other logging methods. We have only to do the following:

1. Write a <START CKPT(T_1,…,T_k)> record to the log, where T_1,…,T_k are all the active transactions, and flush the log.
2. Write to disk all the buffers that are dirty; i.e., they contain one or more changed database elements. Unlike redo logging, we flush all dirty buffers, not just those written by committed transactions.
3. Write an <END CKPT> record to the log, and flush the log.

Notice in connection with point (2) that, because of the flexibility undo/redo logging offers regarding when data reaches disk, we can tolerate the writing to disk of data written by incomplete transactions. Therefore, we can tolerate database elements that are smaller than complete blocks and thus may share buffers.

Example. Consider the following log:

<START T_1>
<T_1,A,4,5>
<START T_2>
<COMMIT T_1>
<T_2,B,9,10>
<START CKPT(T_2)>
<T_2,C,14,15>

<START T_3>
<T_3,D,19,20>
<END CKPT>
<COMMIT T_2>
<COMMIT T_3>

The example is analogous to the example for redo logging. We have changed only the update records, giving them an old value as well as a new value. For simplicity, we have assumed that in each case, the old value is one less than the new value.

When the checkpoint begins, T_2 is identified as the only active transaction. Since this log is an undo/redo log, it is possible that T_2's new B-value 10 has been written to disk, which was not possible under redo logging. However, it is irrelevant whether or not that disk write has occurred. During the checkpoint, we shall surely flush B to disk if it is not already there, since we flush all dirty buffers. Likewise, we shall flush A, written by the committed transaction T_1, if it is not already on disk.

- If the crash occurs at the end of this sequence of events, then T_2 and T_3 are identified as committed transactions. Transaction T_1 is prior to the checkpoint. Since we find the <END CKPT> record on the log, T_1 is correctly assumed to have both completed and had its changes written to disk. We therefore redo both T_2 and T_3, and ignore T_1. However, when we redo a transaction such as T_2, we do not need to look prior to the <START CKPT(T_2)> record, even though T_2 was active at that time, because we know that T_2's changes prior to the start of the checkpoint were flushed to disk during the checkpoint.
- For another instance, suppose the crash occurs just before the <COMMIT T_3> record is written to disk. Then we identify T_2 as committed but T_3 as incomplete. We redo T_2 by setting C to 15 on disk; it is not necessary to set B to 10 since we know that change reached disk before the <END CKPT>. However, unlike the situation with a redo log, we also undo T_3; that is, we set D to 19 on disk. If T_3 had been active at the start of the checkpoint, we would have had to look as far back as the earliest <START T_i> record for all T_i listed in the START CKPT record to find if there were more actions by T_i (now T_2 and T_3) that may have reached disk and need to be undone. For redo recovery, however, we again do not need to look prior to the START CKPT record.
- If the crash occurs prior to the <END CKPT> record, then the last START CKPT record is ignored, and then we do the same as described above.

Protecting Against Media Failures

The log can protect us against system failures, where nothing is lost from disk, but temporary data in main memory is lost. However, more serious failures involve the loss of one or more disks. Theoretically, the database can be recovered after a media failure with the help of the log if

- the disk storing the log is different from the disk(s) containing the database;
- the log is never truncated after creating a checkpoint;
- the log is of type redo or undo/redo, containing the new values of database elements.

However, the log may increase in a faster rate than the database, so it is not a good practice to save the log forever. An archiving system, which we cover next, is needed to enable a database to survive losses involving disk-resident data.

The Archive

To protect against media failures, we are thus led to a solution involving *archiving* — maintaining a copy of the database separate from the database itself. If it were possible to shut down the database for a while, we could make a backup copy on some storage medium such as tape or optical disk and store the copy remote from the database, in some secure location. The backup would preserve the database state as it existed at the time of the backup, and if there were a media failure, the database could be restored to this state.

To advance to a more recent state, we could use the log, provided the log had been preserved since the archive copy was made, and the log itself survived the failure. In order to protect against losing the log, we could transmit a copy of the log, almost as soon as it is created, to the same remote site as the archive. Then, if the log as well as the data is lost, we can use the archive plus remotely stored log to recover, at least up to the point that the log was last transmitted to the remote site.

Since writing an archive is a lengthy process, we try to avoid copying the entire database at each archiving step. Thus, we distinguish between two levels of archiving:

- a *full dump*, in which the entire database is copied;
- an *incremental dump*, in which only those database elements changed since the previous full or incremental dump are copied.

It is also possible to have several levels of dump, with a full dump thought of as a "level 0" dump, and a "level *i*" dump copying everything changed since the last dump at a level less than or equal to *i*. After creating a new level *i* dump, dumps at higher levels can be deleted or ignored during a recovery.

We can restore the database from a full dump and its subsequent incremental dumps in a process much like the way a redo or undo/redo log can be used to repair damage due to a system failure. We copy the full dump back to the database, and then in an earliest-first order, make the changes recorded by the later incremental dumps. In case of multilevel dumps, dumps at levels higher than 0 are processed in an increasing order of levels, and dumps at the same level are processed in chronological order.

We might question the need for an archive, since we have to back up the log in a secure place anyway if we are not to be stuck at the state the database was in when the previous archive was made. While it may not be obvious, the answer lies in the typical rate of change of a large database. While only a small fraction of the database may change in a day, the changes, each of which must be logged, will over the course of a year become much larger than the database itself. If we never archived, then the log could never be truncated, and the cost of storing the log would soon exceed the cost of storing a copy of the database.

Nonquiescent Archiving

The problem with the simple view of archiving described above is that most databases cannot be shut down for the period of time (possibly hours) needed to make a backup copy. We thus need to consider *nonquiescent archiving*, which is analogous to nonquiescent checkpointing. Recall that a nonquiescent checkpoint attempts to make a copy on the disk of the (approximate) database state that existed when the checkpoint started. We can rely on a small portion of the log around the time of the checkpoint to fix up any deviations from that database state, due to the fact that during the checkpoint, new transactions may have started and written to disk.

Similarly, a nonquiescent dump tries to make a copy of the database that existed when the dump began, but database activity may change many database elements on disk during the minutes or hours that the dump takes. If it is necessary to restore the database from the archive, the log entries made during the dump can be used to sort things out and get the database to a consistent state. In other words, a checkpoint gets data

from memory to disk, and the log allows recovery from system failure, whereas a dump gets data from disk to archive, and the archive with the log allows recovery from media failure.

A nonquiescent dump copies the database elements in some fixed order, possibly while those elements are being changed by executing transactions. As a result, the value of a database element that is copied to the archive may or may not be the value that existed when the dump began. As long as the log for the duration of the dump is preserved, the discrepancies can be corrected from the log.

Example. For a very simple example, suppose that our database consists of four elements, A, B, C, and D, which have the values 1 through 4, respectively, when the dump begins. During the dump, A is changed to 5, C is changed to 6, and B is changed to 7. However, the database elements are copied in order, and the sequence of events are the following:

Disk	*Archive*
	Copy A
A := 5	
	Copy B
C := 6	
	Copy C
B := 7	
	Copy D

Then, although the database at the beginning of the dump has values (1,2,3,4), and the database at the end of the dump has values (5,7,6,4), the copy of the database in the archive has values (1,2,6,4), a database state that existed at no time during the dump.

In more detail, the process of making an archive can be broken into the following steps. We assume that the logging method is either redo or undo/redo; an undo log is not suitable for use with archiving (see the discussion after the example for more details).

1. Write a log record <START DUMP>.
2. Perform a checkpoint appropriate for whichever logging method is being used.
3. Perform a full or incremental dump of the data disk(s), as desired, making sure that the copy of the data has reached the secure, remote site.
4. Make sure that enough of the log has been copied to the secure, remote site so that at least the prefix of the log up to and including the checkpoint in item (2) will survive a media failure of the database.
5. Write a log record <END DUMP>.

At the completion of the dump, it is safe to throw away log that is not required according to the recovery rules concerning the checkpoint performed in item (2) above.

Example. Suppose that the changes to the simple database introduced above were caused by two transactions T_1 (which writes A and B) and T_2 (which writes C) that were active when the dump began. The following listing shows a possible undo/redo log of the events during the dump.

<START DUMP>
<START CKPT(T_1,T_2)>
<T_1,A,1,5>
<T_2,C,3,6>
<COMMIT T_2>
<T_1,B,2,7>
<END CKPT>
dump completes
<END DUMP>

Notice that we did not show T_1 committing. It would be unusual that a transaction remained active during the entire time a full dump was in progress, but that possibility doesn't affect the correctness of the recovery method that we discuss next.

Now, we can see why undo log cannot be used with nonquiescent archiving. Suppose a T_3 transaction starts after the <START CKPT(T_1,T_2)> record, which writes A, then B, and then completes, so a <COMMIT T_3> record is written to the log, but only after the <END CKPT> record, i.e., during backup. Since, in case of undo logging, OUTPUT actions can execute at any time after the update record is written to the log, it may happen that A is copied after its value is changed, and B is copied before its value is changed. During recovery, T_3 will be ignored, as its COMMIT record is found in the log. Thus, we get a result as if T_3 had not been executed atomically. Using redo logging, such situations may not happen, because OUTPUT actions may only execute after the COMMIT record is written to the log. This way, either no changes are made on disk (if there is no COMMIT record) or the transaction is redone (if there is a COMMIT record). In case of undo/redo logging, each transaction is undone (if there is no COMMIT record) or redone (if there is a COMMIT record), so there can be no nonatomic behavior.

Recovery Using an Archive and Log

Suppose that a media failure occurs, and we must reconstruct the database from the most recent archive and whatever prefix of the log has reached the remote site and has not been lost in the crash. We perform the following steps:

1. Restore the database from the archive:
    a) Find the most recent full dump and reconstruct the database from it (i.e., copy the archive into the database).
    b) If there are later incremental dumps, modify the database according to each, earliest first. In case of multilevel dumps, apply each dump of each level, beginning with level 1 (in order of levels, and in chronological order within one level).
2. Modify the database using the surviving log. Use the method of recovery appropriate to the log method being used.

Example. Suppose there is a media failure after the dump of the above example completes, and the log survives. Assume, to make the process interesting, that the surviving portion of the log does not include a <COMMIT T_1> record, although it does include the <COMMIT T_2> record. The database is first restored to the values in the archive, which is, for database elements A, B, C, and D, respectively, (1,2,6,4).

Now, we must look at the log. Since T_2 has completed, we redo the step that sets C to 6. In this example, C already had the value 6, but it might be that

- the archive for C was made before T_2 changed C, or
- the archive actually captured a later value of C, which may or may not have been written by a transaction whose commit record survived. Later in the recovery, C will be restored to the value found in the archive if the transaction was committed.

Since T_1 does not have a COMMIT record, we must undo T_1. We use the log records for T_1 to determine that A must be restored to value 1 and B to 2. It happens that they had these values in the archive, but the actual archive value could have been different if the modified A and/or B had been included in the archive. (It depends on the order of the update and the backup of these elements.)

The Logging and Backup System of Oracle Database

The following information comes from the Oracle Database Administrator's Guide.

The Redo Log

After instance failure (system failure of a single instance), Oracle uses the online redo log files to perform automatic recovery of the database. *Instance recovery* occurs as soon as the instance starts up again after it has failed or shut down abnormally. The most crucial structure for recovery operations is the *redo log*, which stores all changes made to the database as they occur. Every instance of an Oracle Database has an associated redo log to protect the database in case of an instance failure. It consists of two parts: online and archived redo log.

An *online redo log* consists of two or more online redo log files, which are filled with *redo records*. A redo record, also called a *redo entry*, is made up of a group of *change vectors*, each of which is a description of a change made to a single block in the database. For example, if you change a salary value in a table containing employee-related data, you generate a redo record containing change vectors that describe changes to the data segment block for the table, the undo segment data block, and the transaction table of the undo segment (see later). Redo records are buffered in a circular fashion in the redo log buffer of the SGA (System Global Area) and are written to one of the redo log files by the Log Writer (LGWR) database background process. (The SGA holds also the buffers for database elements; those buffers are written to disk by the Database Writer background process.) Whenever a transaction is committed, LGWR writes the transaction redo records from the redo log buffer of the SGA to a redo log file, and assigns a *system change number* (SCN) to identify the redo records for each committed transaction. Only when all redo records associated with a given transaction are safely on disk in the online logs is the user process notified that the transaction has been committed. Redo records can also be written to a redo log file before the corresponding transaction is committed. If the redo log buffer fills, or another transaction commits, LGWR flushes all of the redo log entries in the redo log buffer to a redo log file, even though some redo records may not be committed. If necessary, the database can roll back these changes.

The online redo log for a database consists of two or more redo log files. The database requires a minimum of two files to guarantee that one is always available for writing while the other is being archived (if the database is in ARCHIVELOG mode). LGWR writes to redo log files in a circular fashion. When the current redo log file fills, LGWR begins writing to the next available redo log file. When the last available redo log file is filled, LGWR returns to the first redo log file and writes to it, starting the cycle again. Filled redo log files are available to LGWR for reuse depending on whether archiving is enabled. If archiving is disabled (the database is in NOARCHIVELOG mode), a filled redo log file is available after the changes recorded in it have been written to the data files. If archiving is enabled (the database is in ARCHIVELOG mode), a filled redo log file is available to LGWR after the changes recorded in it have been written to the data files *and* the file has been archived. Oracle Database uses only one redo log file at a time to store redo records written from the redo log buffer. The redo log file that LGWR is actively writing to is called the *current* redo log file. Redo log files that are required for instance recovery are called *active* redo log files. Redo log files that are no longer required for instance recovery are called *inactive* redo log files. If you have enabled archiving, then the database cannot reuse or overwrite an active online log file until one of the archiver background processes (ARC*n*) has archived its contents. If archiving is disabled, then when the last redo log file is full, LGWR continues by overwriting the next log file in the sequence when it becomes inactive.

A *log switch* is the point at which the database stops writing to one redo log file and begins writing to another. Normally, a log switch occurs when the current redo log file is completely filled and writing must continue to the next redo log file. However, you can configure log switches to occur at regular intervals, regardless of whether the current redo log file is completely filled. You can also force log switches manually. Oracle Database assigns each redo log file a new *log sequence number* every time a log switch

occurs and LGWR begins writing to it. When the database archives redo log files, the archived log retains its log sequence number. A redo log file that is cycled back for use is given the next available log sequence number. Each online or archived redo log file is uniquely identified by its log sequence number. During crash, instance, or media recovery, the database properly applies redo log files in ascending order by using the log sequence number of the necessary archived and online redo log files.

To protect against a failure involving the redo log itself, Oracle Database allows a *multiplexed* redo log, meaning that two or more identical copies of the redo log can be automatically maintained in separate locations. For the most benefit, these locations should be on separate disks. Even if all copies of the redo log are on the same disk, however, the redundancy can help protect against I/O errors, file corruption, and so on. When redo log files are multiplexed, LGWR concurrently writes the same redo log information to multiple identical redo log files, thereby eliminating a single point of redo log failure.

Oracle Database lets you save filled groups of redo log files to one or more offline destinations, known collectively as the *archived redo log*. The process of turning redo log files into archived redo log files is called *archiving*. This process is only possible if the database is running in ARCHIVELOG mode. You can choose automatic or manual archiving.

When you run your database in NOARCHIVELOG mode, you disable the archiving of the redo log. The database control file indicates that filled redo log files are not required to be archived. Therefore, when a filled redo log file becomes inactive after a log switch, the file is available for reuse by LGWR. NOARCHIVELOG mode protects a database from instance failure but not from media failure. Only the most recent changes made to the database, which are stored in the online redo log files, are available for instance recovery. If a media failure occurs while the database is in NOARCHIVELOG mode, you can only restore the database to the point of the most recent full database backup. You cannot recover transactions subsequent to that backup. In NOARCHIVELOG mode, you cannot perform online tablespace backups, nor can you use online tablespace backups taken earlier while the database was in ARCHIVELOG mode. To restore a database operating in NOARCHIVELOG mode, you can use only whole database backups taken while the database is closed. Therefore, if you decide to operate a database in NOARCHIVELOG mode, take whole database backups at regular, frequent intervals.

When you run a database in ARCHIVELOG mode, you enable the archiving of the redo log. The database control file indicates that a group of filled redo log files cannot be reused by LGWR until the group is archived. A filled group becomes available for archiving immediately after a redo log switch occurs. The archiving of filled groups has these advantages:

- A database backup, together with online and archived redo log files, guarantees that you can recover all committed transactions in the event of an operating system or disk failure.
- If you keep archived logs available, you can use a backup taken while the database is open and in normal system use.
- You can keep a standby database current with its original database by continuously applying the original archived redo log files to the standby.

Oracle Database uses the online redo log only for recovery. However, administrators can query online redo log files through an SQL interface in the Oracle *LogMiner* utility. Redo log files are a useful source of historical information about database activity.

Every Oracle Database has a *control file*, which is a small binary file that records the physical structure of the database. The control file includes:

- the database name,
- names and locations of associated data files and redo log files,
- the timestamp of the database creation,
- the current log sequence number,

- checkpoint information.

The control file must be available for writing by the Oracle Database server whenever the database is open. Without the control file, the database cannot be mounted and recovery is difficult. The control file of an Oracle Database is created at the same time as the database. By default, at least one copy of the control file is created during database creation. On some operating systems the default is to create multiple copies. Every Oracle Database should have at least two control files, each stored on a different physical disk (*multiplexed control file*). If a control file is damaged due to a disk failure, the associated instance must be shut down. Once the disk drive is repaired, the damaged control file can be restored using the intact copy of the control file from the other disk, and the instance can be restarted. In this case, no media recovery is required.

Undo Management

Oracle uses a special combination of undo and redo logging. As we have seen, information for redo recovery (the new values of database blocks) is stored in the redo log. Information for undo recovery, however, are stored in one or more *undo tablespaces* by default (or in *rollback segments* placed in other tablespaces; see later). This means that Oracle Database stores undo data inside the database rather than in external logs. Undo data is stored in blocks that are updated just like data blocks, with changes to these blocks generating redo records. In this way, Oracle Database can efficiently access undo data without needing to read external logs. Undo tablespaces record the old values of data that was changed by each transaction (whether or not committed). Oracle uses undo data to roll back an active transaction, recover a terminated transaction, provide read consistency, and perform some logical flashback operations.

An undo tablespace consists of *undo segments*, which consist of *undo records* or *undo entries*. The contents of an undo entry include the address of changed data column(s), the transaction operation performing the change, an SQL statement that undoes the effect of the change, and the old value(s) of each changed column. Undo entries are always written to disk before the corresponding modified data reach disk. Undo entries for each transaction are linked so that they can be located easily for undo activities.

Each undo segment of an undo tablespace has a corresponding *transaction table*, which holds the transaction identifiers of the transactions using the undo segment. The transaction table is made up of a fixed number of slots or entries. This figure depends on the size of a data block, which is defined by the operating system. Each of these slots is assigned to a transaction and contains information about that action. The slots are initially used in order but are reused in a round-robin fashion with one exception: a slot referencing an uncommitted transaction will not be reused. It is possible to fill up all slots with active transactions. If this occurs, the transaction waits until a slot becomes available.

After a transaction is committed, undo data is no longer needed for rollback or transaction recovery purposes. However, for consistent read purposes, long-running queries may require this old undo information for producing older images of data blocks (see chapter titled “Concurrency Control in Oracle”). Furthermore, the success of several Oracle Flashback features can also depend upon the availability of older undo information. For these reasons, it is desirable to retain the old undo information for as long as possible.

An auto-extending undo tablespace named UNDOTBS1 is automatically created when you create the database with Database Configuration Assistant (DBCA). You can also create an undo tablespace explicitly, using the CREATE DATABASE or CREATE UNDO TABLESPACE statement. When the database instance starts, the database automatically selects the first available undo tablespace. If no undo tablespace is available, then the instance starts without an undo tablespace and stores undo records in the SYSTEM tablespace. This is not recommended, and an alert message is written to the alert log file to warn that the system is running without an undo tablespace. If the database contains multiple undo tablespaces, then you can optionally specify at startup that you want to use a specific undo tablespace. This is done by setting the UNDO_TABLESPACE initialization parameter.

Instance Recovery Phases

As changes are made to the undo segments, these changes are also written to the online redo log. It is because undo tablespace is part of the database just like other tablespaces. As a result, the online redo log always contains the undo data for permanent objects. This means that every change to the database implies the creation of an undo entry with the old value of the changed column(s), a log record with the new value of the data block containing the modified data, and another log record with the new value of the data block containing the undo entry.

The first phase of instance recovery is called *cache recovery* or *rolling forward*, and involves reapplying all of the changes recorded in the online redo log to the data files. It is enough to reconstruct changes made after the most recent checkpoint. The checkpoint position guarantees that every committed change with an SCN lower than the checkpoint SCN is saved to the data files. Checkpoints occur in a variety of situations. For example, when the Database Writer process writes dirty buffers, it advances the checkpoint position. The resulting database state after a cache recovery is very likely to be inconsistent. After the roll forward, any changes that were not committed must be undone. Because rollback data is recorded in the online redo log, rolling forward also regenerates the corresponding undo segments. Oracle Database applies undo blocks to roll back uncommitted changes in data blocks that were written before the failure or introduced during cache recovery. This phase is called *rolling back* or *transaction recovery*.

Undo Management Modes

The database can run in *automatic* or *manual undo management mode*. With automatic undo management, the database automatically manages undo segments in undo tablespaces, and no user intervention is required. Automatic undo management is the default mode for a newly installed database. In manual mode, undo space is managed through *rollback segments* (user-managed undo segments), and no undo tablespace is used. Space management for rollback segments is complex and requires hard work from the DBA.

Backup and Recovery

The focus in Oracle Database backup and recovery is on the physical backup of database files, which permits you to reconstruct your database. RMAN, a command-line tool, is the method preferred by Oracle for efficiently backing up and recovering your Oracle database. The files protected by the backup and recovery facilities built into RMAN include data files, control files, server parameter files, and archived redo log files. With these files you can reconstruct your database. RMAN is designed to work intimately with the server, providing block-level corruption detection during backup and restore. RMAN optimizes performance and space consumption during backup with file multiplexing and backup set compression, and integrates with leading tape and storage media products. The backup mechanisms work at the physical level to protect against file damage, such as the accidental deletion of a data file or the failure of a disk drive. RMAN can also be used to perform point-in-time recovery to recover from logical failures when other techniques such as flashback cannot be used.

In NOARCHIVELOG mode, the filled redo log groups that become inactive can be reused. This mode protects the database against instance failure, but not against media failure. In ARCHIVELOG mode, filled groups of redo logs are archived. This mode protects the database from both instance and media failure, but may require additional hardware resources.

A *full backup* of a data file includes all used blocks of the data file. An *incremental backup* copies only those blocks in a data file that change between backups. A *level 0 incremental backup*, which copies all blocks in the data file, is used as a starting point for an incremental backup strategy. A *level 1 incremental backup* copies only images of blocks that have changed since the previous level 0 or level 1 incremental

backup. Level 1 backups can be *cumulative*, in which case all blocks changed since the most recent level 0 backup are included, or *differential*, in which case only blocks changed since the most recent level 0 or level 1 incremental backup are included. A typical incremental strategy makes level 1 backups at regular intervals such as once each day. During recovery, RMAN will automatically apply both incremental backups and redo logs as required, to recover the database to the exact point in time desired.

A backup is either *consistent* or *inconsistent*. To make a consistent backup, your database must have been shut down cleanly and remain closed for the duration of the backup. All committed changes are written to the data files during the shutdown process, so the data files are in a transaction-consistent state. When you restore your data files from a consistent backup, you can open the database immediately. If the database is in ARCHIVELOG mode, then you can make inconsistent backups that are recoverable using archived redo log files. Open database backups are inconsistent because the online redo log files contain changes not yet applied to the data files. The online redo log files must be archived and then backed up with the data files to ensure recoverability. Despite the name, an inconsistent backup is as robust a form of backup as a consistent backup. The advantage of making inconsistent backups is that you can back up your database while the database is open for updates.

If you restore the archived redo log files and data files, then you must perform *media recovery* before you can open the database. Any database transactions in the archived redo log files not reflected in the data files are applied to the data files, bringing them to a transaction-consistent state before the database is opened. Media recovery requires a control file, data files (typically restored from backup), and online and archived redo log files containing changes since the time the data files were backed up. Media recovery is most often used to recover from media failure, such as the loss of a file or disk, or a user error, such as the deletion of the contents of a table. Media recovery can be a *complete recovery* or a *point-in-time recovery*. Complete recovery can apply to individual data files, tablespaces, or the entire database. Point-in-time recovery applies to the whole database (and also sometimes to individual tablespaces, with automation help from RMAN). In a complete recovery, you restore backup data files and apply all changes from the archived and online redo log files to the data files. The database is returned to its state at the time of failure and can be opened with no loss of data. In a point-in-time recovery, you return a database to its contents at a user-selected time in the past. You restore a backup of data files created before the target time and a complete set of archived redo log files from backup creation through the target time. Recovery applies changes between the backup time and the target time to the data files. All changes after the target time are discarded.